CSCI 3753: Operating Systems

Week 7
October 10th, 2025

University of Colorado Boulder

Announcements

- PS2 due Sunday at midnight!
- Quiz 7 due today at midnight
- Midterm is next week!
- As always recitation materials are stored here

On-Going

- PA4
  - Create a shared array
- PA5
  - Create a DNS resolver
- PA6
  - Combine these to create a multi-threaded DNS resolver

Bounded Buffer Problem!!

Machine Components, Bootloader, Kernel Mode and System Calls (Chapter 2)

- What are the roles of BIOS/UEFI, boot loader, and kernel in the startup sequence?
- Why do we need kernel mode vs user mode? What would happen if there were only one mode?
- How does a system call differ from a regular function call?
- Can you give examples of system calls in Unix/Linux?
- What is the purpose of the trap table?

Loadable Kernel Modules, Bus Controllers, DMA, Device Drivers (Chapters 11+12)

- Why do we use LKMs instead of compiling everything directly into the kernel?
- What is the difference between kernel-space and user-space drivers?
- How does DMA improve I/O performance?
- What are interrupts and how do they relate to device drivers?
- What happens when a device driver misbehaves?

Processes and Interprocess Communication (Chapter 3)

- What are the main components of a process (address space, registers, PCB, etc.)?
- How does process creation work (fork/exec model in Unix)?
- What are the main IPC mechanisms (pipes, message queues, shared memory, sockets)?
- Why do we need synchronization when using shared memory?
- How is a process different from a program?

Threads, Thread Safety and Reentrant Code (Chapter 4)

- How do threads differ from processes?
- Why is shared memory both an advantage and a danger in threads?
- What makes a function thread-safe?
- What does “reentrant” mean, and are reentrant functions always thread-safe?
- Why do many OS-level services (like signal handling or printf) need to consider thread safety?

Synchronization (Chapters 6+7)

- What is a race condition? Can you describe one example?
- What are the three requirements for a critical section solution (mutual exclusion, progress, bounded waiting)?
- Why are busy-waiting and spinlocks inefficient? When might they still be used?
- How do hardware-level atomic instructions (test-and-set, compare-and-swap) help?

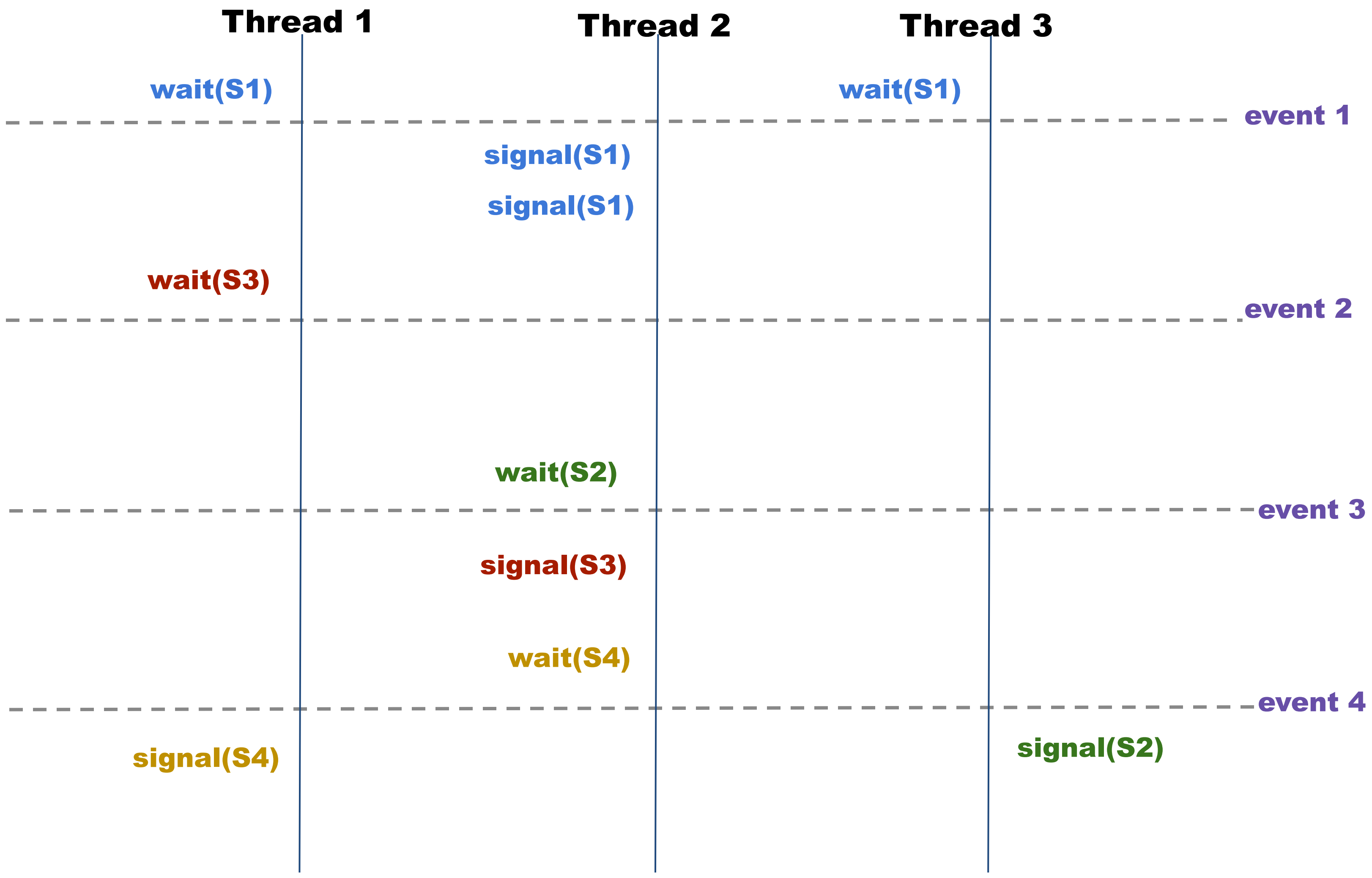
Mutexes, Semaphores, Monitors and Condition Variables (Chapters 5,6+7)

- How does a mutex differ from a semaphore?
- What does wait() and signal() do in a semaphore context?
- Why do monitors simplify synchronization compared to semaphores?
- What is a condition variable used for inside a monitor?
- Can semaphores cause deadlock? When?

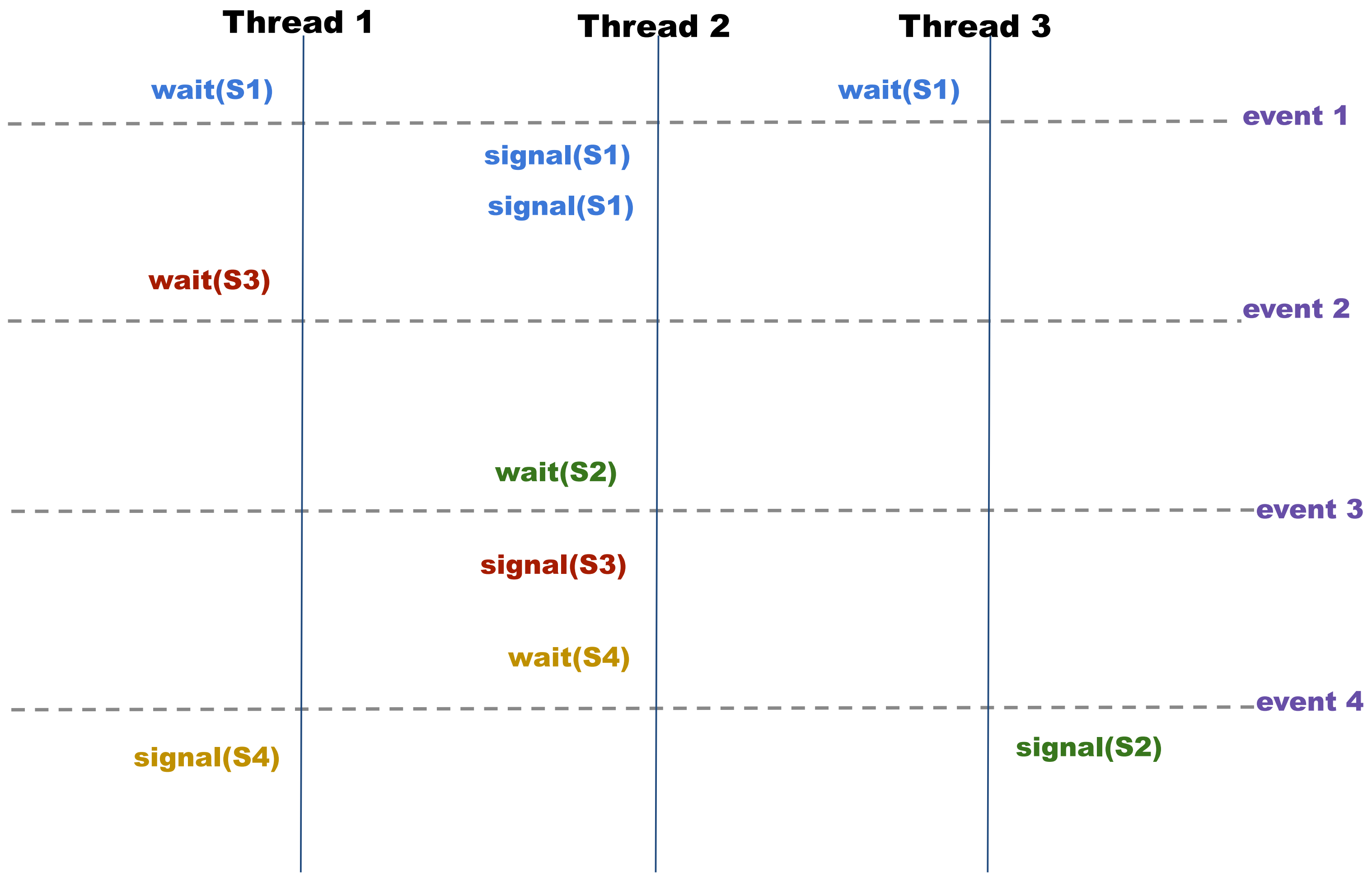
Deadlock - Conditions, Detection, and Avoidance (Chapters 7+8)

- What are the four necessary conditions for deadlock?
- How does a resource-allocation graph help us detect deadlocks?
- What’s the difference between avoidance (Banker’s algorithm) and prevention?
- Is it ever acceptable to let deadlock occur? (Why might Linux do this?)
- How can ordering resource requests avoid circular wait?

Semaphore Question

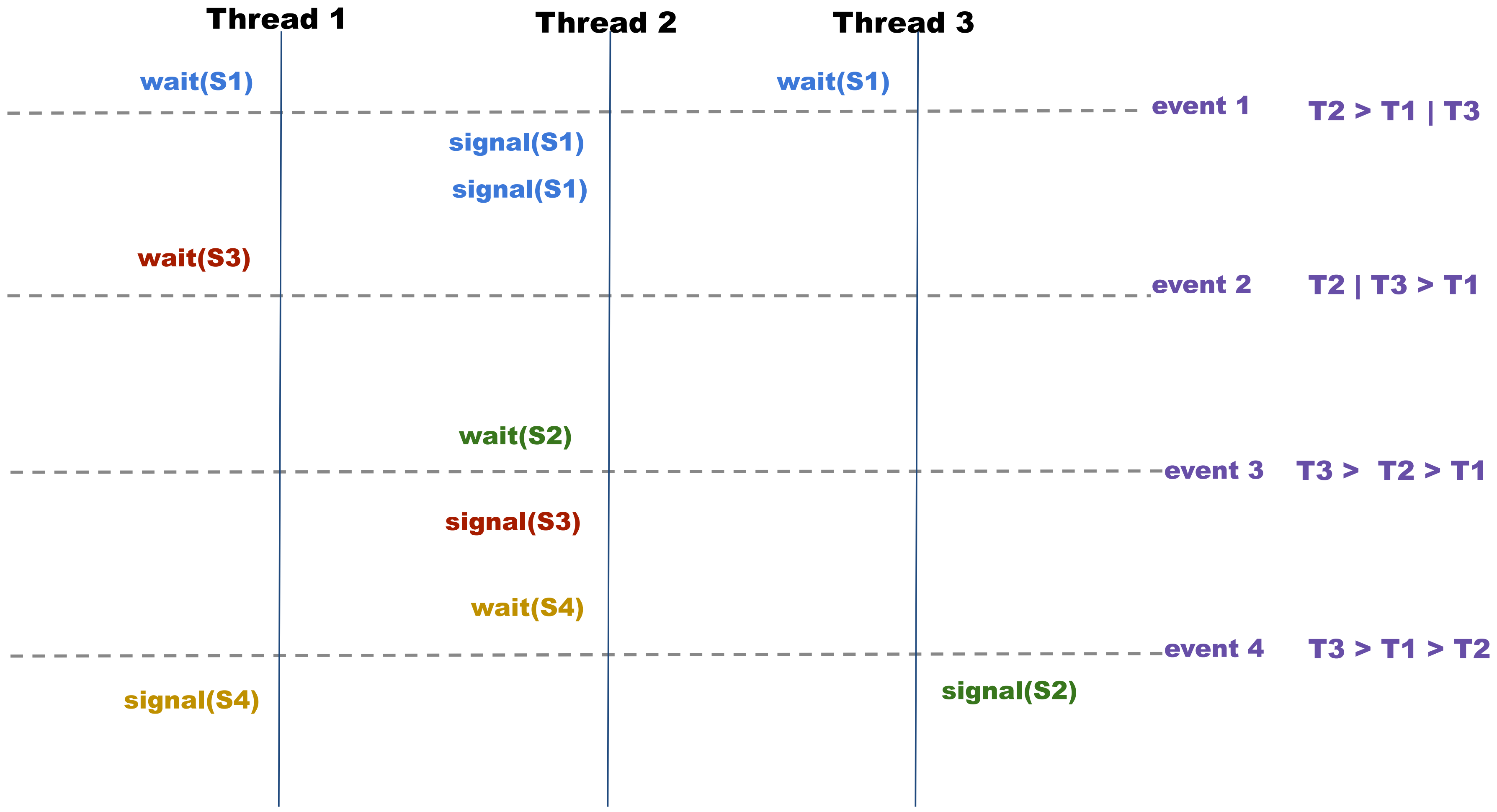


Semaphore Question

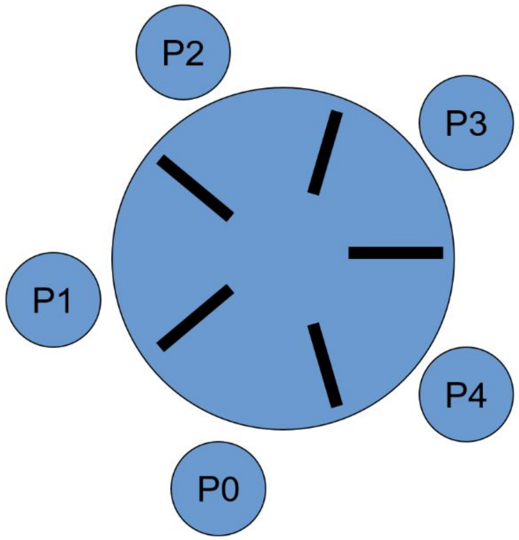


In what order order do these threads execute each event?

Semaphore Question



Philosophers Dining Problem



```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }

   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }

   Putdown(int i) {
         state[i] = thinking;
         test((i+1)%5);
         test((i-1)%5);
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }
} // end of monitor
```

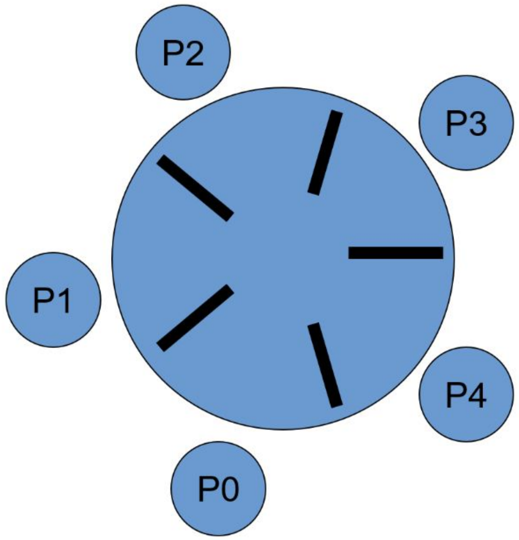
1 - Initially, philosophers P0 and P3 arrive first and start eating (call to Pickup(i) is successful). P4 arrives next and sets its state to hungry, blocking on its condition variable. If P0 finishes eating, who will be next to eat?

2 - Assume the philosopher from your answer in part a is now eating. Next, P1 will arrive and set its state to hungry, is P1 able to begin eating?

3 - Now P3 will finish eating, who will be next to eat?

4 - Assume P0 is gluttonous and never stops eating, identify any issues with our table

Philosophers Dining Problem



```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }
```

```
Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }

    init() {
        for i = 0 to 4
            state[i] = thinking;
    }
}   // end of monitor
```

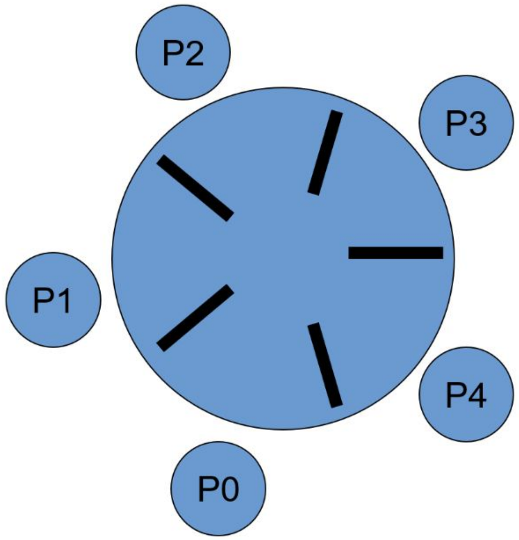
1 - Initially, philosophers P0 and P3 arrive first and start eating (call to Pickup(i) is successful). P4 arrives next and sets its state to hungry, blocking on its condition variable. If P0 finishes eating, who will be next to eat? **P0**

2 - Assume the philosopher from your answer in part a is now eating. Next, P1 will arrive and set its state to hungry, is P1 able to begin eating?

3 - Now P3 will finish eating, who will be next to eat?

4 - Assume P0 is gluttonous and never stops eating, identify any issues with our table

Philosophers Dining Problem



```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }
```

```
Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }

    init() {
        for i = 0 to 4
            state[i] = thinking;
    }
}   // end of monitor
```

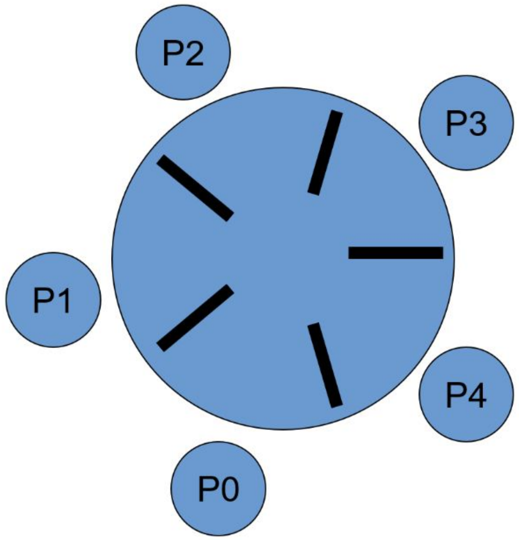
1 - Initially, philosophers P0 and P3 arrive first and start eating (call to Pickup(i) is successful). P4 arrives next and sets its state to hungry, blocking on its condition variable. If P0 finishes eating, who will be next to eat? **P0**

2 - Assume the philosopher from your answer in part a is now eating. Next, P1 will arrive and set its state to hungry, is P1 able to begin eating? **NO**

3 - Now P3 will finish eating, who will be next to eat?

4 - Assume P0 is gluttonous and never stops eating, identify any issues with our table

Philosophers Dining Problem



```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }

Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }

    init() {
        for i = 0 to 4
            state[i] = thinking;
    }
}   // end of monitor
```

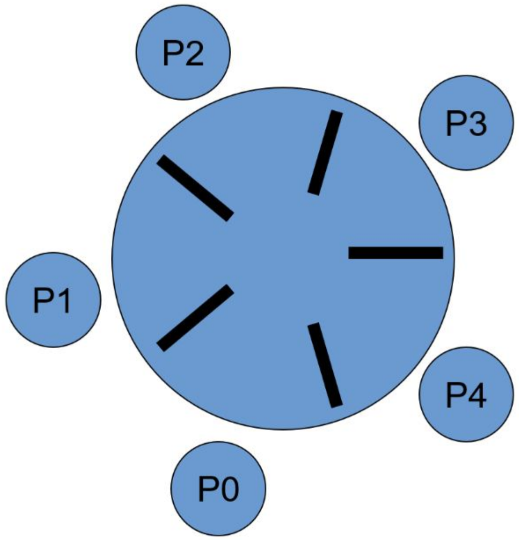
1 - Initially, philosophers P0 and P3 arrive first and start eating (call to Pickup(i) is successful). P4 arrives next and sets its state to hungry, blocking on its condition variable. If P0 finishes eating, who will be next to eat? **P0**

2 - Assume the philosopher from your answer in part a is now eating. Next, P1 will arrive and set its state to hungry, is P1 able to begin eating? **NO**

3 - Now P3 will finish eating, who will be next to eat? **P3**

4 - Assume P0 is gluttonous and never stops eating, identify any issues with our table

Philosophers Dining Problem



```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }

    Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }

    init() {
        for i = 0 to 4
            state[i] = thinking;
    }
}   // end of monitor
```

1 - Initially, philosophers P0 and P3 arrive first and start eating (call to Pickup(i) is successful). P4 arrives next and sets its state to hungry, blocking on its condition variable. If P0 finishes eating, who will be next to eat? **P0**

2 - Assume the philosopher from your answer in part a is now eating. Next, P1 will arrive and set its state to hungry, is P1 able to begin eating? **NO**

3 - Now P3 will finish eating, who will be next to eat?

P3

4 - Assume P0 is gluttonous and never stops eating, identify any issues with our table

Philosophers are starving

Resource Question

A system has 13 identical resources and N processes competing for them. Each process can request at most 4 resources.

What is the maximum value of N we can accommodate such that no deadlock can ever occur?

Resource Question

A system has 13 identical resources and N processes competing for them. Each process can request at most 4 resources.

What is the maximum value of N we can accommodate such that no deadlock can ever occur?

4 processes

Resource Question

A system has 13 identical resources and N processes competing for them. Each process can request at most 4 resources.

What is the maximum value of N we can accommodate such that no deadlock can ever occur?

4 processes

Starting at N=5, we could have:

3 + 3 + 3 + 3 +1 == DEADLOCK!

Resource Question

A system has 13 identical resources and N processes competing for them. Each process can request at most 4 resources.

What is the maximum value of N we can accommodate such that no deadlock can ever occur?

4 processes

Starting at N=5, we could have:

3 + 3 + 3 + 3 +1 == DEADLOCK!

But with N=4:

3 + 3 + 3 + 4 == NO DEADLOCK